|  |
| --- |
| Intelligent Recommendation Service |
|  |
| Implementation Report |

|  |
| --- |
| By Brian Davis  1-22-2021 |

Table of Contents

[Introduction 2](#_Toc61790843)

[Problem Analysis 2](#_Toc61790844)

[Solution Requirements 2](#_Toc61790845)

[Functional Requirements 2](#_Toc61790846)

[Non-functional Requirements 3](#_Toc61790847)

[Implementation of Solution 3](#_Toc61790848)

[Program Execution 4](#_Toc61790849)

[Personal Reflection 5](#_Toc61790850)

[Dataset Loading 5](#_Toc61790851)

[Similarity Metrics 5](#_Toc61790852)

[Generating Recommendations 5](#_Toc61790853)

[Main Function 6](#_Toc61790854)

[Evaluation 6](#_Toc61790855)

[Conclusion 6](#_Toc61790856)

[References 7](#_Toc61790857)

[Appendix 8](#_Toc61790858)

[Program Structure Flowchart 8](#_Toc61790859)

[System Architectural Diagram 8](#_Toc61790860)

[Use Case Textual Diagram 8](#_Toc61790861)

[Class Diagram 8](#_Toc61790862)

[Pseudocode 9](#_Toc61790863)

[Requirement Testing Matrix 10](#_Toc61790864)

[Evaluation Plots 10](#_Toc61790865)

# Introduction

With the growing emergence of online content as a service, there is need for the implementation of an Intelligent Recommendation Service, which will allow the company to increase their revenue, by being able to recommend their products to customers using gained knowledge regarding their interests. The program outlined in this report aims to meet the goals and exceed client expectations with capability to enable the comparison of song features through a wide range of similarity metrics and the ability to provide recommendations based on a target entry. The system does this through successful reading of the file structure, appropriate assignment through suitable data structures and implementation of suitable libraries, alongside key comparison of suitable features collected from the provided data.

# Problem Analysis

To create the solution, the problem needs to be broken down into steps. The steps needed for this implementation are as follows.

1. Reading of the music dataset file
2. Searching artists and songs
3. Computing Similarity between artists / music tracks by ID
4. Generating recommendations to a user based on their target ID input

To accomplish these steps, it is important to analyse the problem before deciding on the approach to take. There are also a variety of other things to consider when thinking about the best way to approach the problem. One of the problems involved in a Recommendation Service is new users. It’s the aim of the service to provide something that draws in new users, but the problem of recommending songs to a new user is that their tastes in music are not existent on the system. To understand the taste of the user, it is important for the program to be subjective and not recommend many things to a user until there is more data to make use of, so the algorithm can be accurate in its assumptions. It can also be a good idea to recommend songs to a user to see what the response would be.

The steps defined above showcase the overall direction of the implementation of the system, and the stages that must be considered when planning the implementation documents. Notably, there is no avenue for a user, new or existing, to search for songs within the dataset. There needs to be an avenue to allow the user to be able to search by name or closest match. The metrics to be created for the similarity scoring need to be tested for suitability of purpose, as not all metrics will be useful, but having some variety to choose from is never a bad option either. Generating recommendations for the user also depends on their target, and there are a few different types of recommendation styles that can be used. In this case, the program uses Content Based Filtering, which takes a target from the user and generates recommendations based on this target in real time.

# Solution Requirements

This section outlines the characteristics of the solution, and how these characteristics enable the program to meet the needs of the stakeholders and the business.

## Functional Requirements

When considering the functional requirements, there are a few aspects that are key for this program. The most important of these is the **user requirements**, which can be showcased in a use case diagram (see appendix). This style of diagram helps to show what the program does and how this meets the requirements of the user. This highlights what the program does and how it meets what the user would be expecting when they are using the program.

Another important requirement to consider is the **system requirements**. For this program, the system requirements are small. The main requirement to run the program is a system that can access Python and run a python notebook in the environment of their choice. This could be through Jupyter Notebook or Visual Studio Code using the python add-on, as the system uses a simple UI for a simple program with effective ease of use, including some modular flexibility. An important thing to note here is that the modules that need to be imported must remain in the same folder as the solution, as not to break functionality.

## Non-functional Requirements

The non-functional requirements aim to define the system behaviour. This section discusses these requirements and how they must be met when creating the solution.

The non-functional requirements have been listed in this section. The first is **Usability**, which refers to how easy the program is to use for an end user. The program will use a simple UI, which will require input from the user in the form of text entry. The use of this simple UI for the program input makes the solution efficient, intuitive and maintains a low perceived workload. There are plenty of text outputs for the user so that they understand and can easily follow along and input the relevant response to successfully proceed within the program. Any errors are relayed back to the user in an understandable manner.

Continuing, the next requirement is **Supportability**, which refers to how easy it is to update the codebase when required. The system will make use of Object-Oriented Programming principles with the aim to make the codebase as compact as possible. Making use of OOP allows for the program to eliminate need for unnecessary code duplication in places by making use of Inheritance to share methods from one class to another. Good use of code structure that follows coding standards and contains useful comments will allow a future developer to be able to refresh and update the code when and if necessary.

Another consideration is **Appropriateness**, which is the suitability of the program for its intended purpose. This requirement is simply a measure of how well the program meets its intended purpose, which is to recommend songs or artists to a user based on what they already like. Successful implementation of the program will allow it to meet this requirement without any issue.

The program also needs to be **Reliable**. This involves the effective use of exception handling, which makes sure that the program doesn’t experience any crashes if the user inputs a value that would normally create an error, such as when a data frame is called when it is not yet instantiated. The program will therefore make effective use of built-in exception handling to catch all exceptions in the program, and instead of crashing, will print a message to the user, and then re-run the section of the program that was interrupted due to an error if appropriate.

The last non-functional requirement is **Performance**. The program runs through a single execution that calls multiple modules through various sections of the UI. This makes performance of the program fast; the lightweight nature of the implementation will allow the program to run fast and snappy when being used. While there is no need for the program to run fast, the nature of the implementation makes a fast-running program easy to accomplish. The slowest aspect of the program is when the user enters their target, and the program calculates the metric of the target against the library to find recommendations.

# Implementation of Solution

This section outlines the steps taken to accomplish program implementation. More thorough discussion regarding the execution of the program can be found in later sections.

The implementation of the program was an incremental process, which involved working in steps to complete sections of the codebase and returning if any issues came up on previously completed sections. Use of GitHub for saving work and creating issues helped to gauge progress. To accomplish the file loading, the Python library Pandas was used within a file loader class. Using this library meant that all encoding and splitting of data was done automatically using a comma as the delimiter.

Three Classes are used to split up the data as Artist, Song and Extras, where the artist name is included in the Artist class, the song features used for comparison alongside the song name are within the Song class, and the extra features of the data that are not used are within the Extras class. A Track class uses Multiple Inheritance to inherit all the keyword arguments and methods from these three classes and uses them to create a combined class for the entire dataset. User searching and the search function are included in final implementation, which helps if the user doesn’t know the necessary ID. As the program takes an ID number as the input, if the user doesn’t know the ID, then the search function will help them to find it, to pull the information for comparison or recommendation.

A method was created within the Track class to return a data frame to be used later for calculations. Functionality to present the number of successful matches within artist and song searches allows user feedback regarding the accuracy of their search, and then presents them with a box asking them if they want to view the results. This is done so that when a user conducts a search, the program doesn’t simply throw all the results at the user without some sort of numeric feedback beforehand. The user can reject the search results and move on in the program if they wish. Searching for a song can be done infinitely until the user has had enough and wishes to proceed, this adds flexibility and some modularity to the solution.

Creation of the metrics was done through method creation and use of the NumPy and SciPy libraries to extract the metric functions needed, which were then called using dot notation. The overall functionality of the similarity functions includes error checking to avoid incorrect input, when the same numerical ID is entered, the program will output 1 for comparison, alongside the ability to reset the program if a feature was entered that couldn’t be found. Once all metric functions were created and checked for errors, the recommendation class was created.

The recommendation class was created using inherited properties from the Similarity metric class which housed the metrics to be used. A choice method is also created to define the metric to be used, as some of the metrics require the sorted results to be reversed and some don’t, so this needs to be specified in the code to avoid presenting false results to the user. To keep the data to a similar scale, a scaler function is used called a MinMaxScaler which normalises all the values within the data before transformation occurs to turn the data into an array. This is done because doing calculations on a data frame took a long time, and the array turned out to be much shorter in this respect.

There is the issue of the program returning the same song / artist/s that are being sent as the target, which is corrected by removing the target from the library before any recommendations can be calculated. The algorithm is chosen by the user and then the program takes the target and loops through the library, using this metric on the target v every individual vector within the array before returning a set of the n closest results to the target, where the n value is chosen by the user as a multiple of 5, with 5 being the lowest value accepted. These results are sorted by the lowest value to the highest before they are shown to the user. The K Nearest Neighbor algorithm is used as a sole method due to the inability to use the full classifier for this task. The KNN algorithm is used as the final method as another way to get results that doesn’t simply use the metrics created and uses the algorithm from a library. It aims to be not only another avenue for results, but also as an avenue for comparisons of result accuracy in the Evaluation section.

# Program Execution

This section discusses the implementation and execution of the program through the main notebook. The section focuses on the choices made and the structural decisions when deciding upon program flow and flexibility.

The program executes through the main notebook by calling the Main class. This class is created with methods that call methods from the other classes within the two imported modules. The program is easy to use with an intuitive UI that guides the user through their selections and presents suitable feedback when incorrect entries are identified. The program will attempt to recover when errors are identified in inputs, but where this is not possible, the program will need to be started again by re-running the code block. The use of methods within the classes aims to reduce code duplication at every possible opportunity.

A flow chart is designed for an easy-to-read diagram that showcases the overall flow and flexibility of the program, dependant on the instruction of the user. This flow chart can be found within the appendix. It was important to create a program that closely matched the flow chart, and it is successful in that regard. A System Architectural Diagram is located within the appendix and shows the design for the programs intended functionality. There is also a Class diagram within the appendix which is required to show the relationships between the classes that have been created for the solution.

When creating the main program, it was important to give the user a choice of what they would like to do. For this, markdown is used to guide the user within the notebook, alongside the use of printed output to guide the user within the program. A main function is present, but code blocks are available toward the bottom of the notebook, to allow a modular execution if this direction is preferred. Execution of the program had some initial problems, but these were solved with extra error handling. Pseudo code for the main function can also be found in the appendix, alongside pseudo code for all created classes and methods within the solution.

The relationship between the modules is also of note. The program first uses the load dataset module to get the datasets, after which the program will then call the functions defined in the similarity module. While there is no real direct relationship between the modules, without the load module, the program can’t function as the code will falter without the defined dictionaries. The strongest relationship is with the functions found within the similarity module, which all work in tangent to allow the user to search for artists and use of metrics to get accurate results printed to them.

# Personal Reflection

This section provides an overview of the main issues found within the program during implementation and outlines the ways that they were corrected where necessary. The section mainly discusses the implementation of the two modules for the program, and the implementation of the main notebook.

## Dataset Loading

Loading the data is the first part of the implementation. Working with the file was seamless whilst making use of Pandas. The use of zip allowed the data to be added to the Track class as a class-based instance, which essentially made the file a dictionary with additional functionality. Indexing is done automatically by Pandas. IDs are present in searches by the user for songs or artists.

## Similarity Metrics

The program uses 5 similarity metrics to run comparisons on features and generate recommendations. These metrics are Euclidean, Manhattan, Pearson Correlation, Cosine and Jaccard. When working with these metrics, it was increasingly important that rigorous testing was conducted to make sure that there were no issues regarding inputs, outputs, and calculations of the results. The premise of comparisons is a value against another value, whilst recommendations work by taking a target set of values and comparing them to all other values for each other item in the library. A problem that arose in using two features of shape (1, ) is that Pearson Correlation shouldn’t be used in this respect. This is because a correlation requires at least two x values and two y values to make a prediction of correlation and output an accurate result. When using correlation for recommendations however, no issues were discovered.

Euclidean and Manhattan are successful for comparing and recommending items for a user, as these algorithms work by taking the first item from the second and applying some other math to the results to make them differ from one another. Comparisons done with these two metrics shared the same result, but for recommendations, these results were no longer identical in score. Euclidean and Manhattan are accurate across all values, leading to a successful implementation of these metrics. The same can be said for the metrics of Cosine and Jaccard, where the issues found when working with these metrics were small, although Jaccard is not a suitable metric for this task, as most of the time the output is 0 or 1, and is not very informative, due to what Jaccard is aiming to tell the user about the values they are inputting.

## Generating Recommendations

When deciding on the best approach for generating n recommendations, the first initial thought was to adapt the solution used when the program compares all features from an item against another item. Creating a target variable and taking a feature and looping through all items and doing the calculation on this value against all other items was successful, but the results were not as accurate as they could be. To further expand and improve upon this solution, it was decided to take the id number of an item and loop through all other items and do the metric calculation of all features of an item against all other items instead to generate a more accurate recommendation engine.

The use of a data frame to implement this was initially very slow, due to the dimensionality of the data. Instead, using a scaler to normalise all values into an array, and then calculate similarity proved to be incredibly fast in comparison, with the slowest result taking 10 seconds using Pearson Correlation. Once these results were compiled and added to a list, they were sorted by their scores and then the IDs of the ordered results were used to pull the names and print them to the user. The use of the class-based list from file loading proved incredibly useful for this. The loop to print results to the user will then terminate when the number of printed results reaches the value of the n that the user enters.

## Main Function

The aim of the final implementation of the UI was to create an intuitive and user-friendly UI. The structure of this UI had to follow the flow chart exactly and this was successful. Implementation of the UI didn’t have any critical issues, as the use of OOP principles made the UI very easy to code within the Main class, making use of methods to avoid code duplication where this was feasible to do so. While loops are used in two instances within the UI method. One of these is for when the user says they don’t know the IDs they need to make use of the program, which contained an incrementor that would terminate the program if the input was incorrect more than 3 times.

The other instance was a while loop that would allow the user to search for artists or songs until they enter ‘continue’ into the input box. An issue not corrected for the search results is the abundance of results being printed all at one rather than in increments. This is only really an issue when the user searches for a song rather than when they search for an artist, due to the way the song search was implemented, which looks for words in song titles, and entering more than one word would result in a lot of results due to the weakness of the song search implementation. A results box is part of the UI to avoid all results being printed without the user asking to see them, but the issues with the song search functionality remain unresolved.

Input boxes can be prone to spaces being entered. This was avoided by using the strip() command for inputs. This would take all leading and trailing spaces away from any inputs. Capitalisation is used for string-based inputs as the names of features are capitalised within the data frame, so even if the user enters all upper case or all lower case, their input will still be correct unless the spelling is incorrect.

# Evaluation

To decide which is the best metric in terms of the overall recommendation accuracy, there is need to evaluate the recommendations given by each metric against the others. Doing this wouldn’t be ideal, so instead a dummy set is created with a dummy target to see how each metric decides based on the values of the target, in what order to recommend items. Looking at the results, it is possible to see that all metrics rank the dummy items similarly except for Jaccard, where the accuracy of each other metrics ranking against Jaccard is 10%. This means that the metrics other than Jaccard have a similar accuracy when we define a dummy set of items.

Plots help to show the overall mean of the item values and how close to the target mean of values these recommendations are. Most of the time, the target and recommendations have very similar mean of item values, except when looking at Jaccard, which has a varied mix considering the mean of item values. Overall, there is some confidence to suggest that Euclidean and Manhattan output a similar order to their recommendations, as do Pearson and Cosine. This is simply due to the way these algorithms work, which makes sense that they align to each other more than they do to the others when considering ranking of the items based on the target, but there are some outliers depending on what the target is. All Plots exploring this can be found within the appendix, towards the bottom of the main notebook.

# Conclusion

This report showcases the steps taken to create a final runnable solution for a potential user. The main aims for the project have been fulfilled and the searching feature is included as it is necessary to allow a user to find the target ID, they want to use to get recommendations. Overall, the program has reached a suitable state that all parties can be happy with. The use of libraries helps to enhance the capabilities of the program by reducing execution time for file loading and recommendation calculations in real time.

# References

Harris, C.R., Millman, K.J., van der Walt, S.J. et al. Array programming with NumPy. Nature 585, 357–362 (2020). DOI: [0.1038/s41586-020-2649-2](https://doi.org/10.1038/s41586-020-2649-2).

Van Rossum, G. (2020). *The Python Library Reference release 3.8.2*. Python Software Foundation.

Schedl, M., Zamani, H., Chen, C.-W., Deldjoo, Y., & Elahi, M. (2018). Current challenges and visions in music recommender systems research. *International Journal of Multimedia Information Retrieval*, *7*(2), 95–116. <https://doi.org/10.1007/s13735-018-0154-2>

*What is a Flowchart*. (n.d.). Lucidchart. Retrieved 28th November 2020, from <https://www.lucidchart.com/pages/what-is-a-flowchart-tutorial>

*Guthrie, G. (2020, August 2). Everything you need to know about architectural diagrams (and how to draw one). Cacoo.* [*https://cacoo.com/blog/everything-you-need-to-know-about-architectural-diagrams-and-how-to-draw-one/*](https://cacoo.com/blog/everything-you-need-to-know-about-architectural-diagrams-and-how-to-draw-one/)

*Functional vs Non-Functional Requirements: The Definitive Guide*. (2019, September 30). QRA Corp. <https://qracorp.com/functional-vs-non-functional-requirements/>

# Appendix

### Program Structure Flowchart

The flowchart that shows the overall expected flow of the program is found here, provided below. A small label has been added to the right side of the flow-chart, so that a reader can understand what ID refers to here if they are not sure.

### System Architectural Diagram

### Use Case Textual Diagram

|  |  |  |
| --- | --- | --- |
| Actors | Use Cases | Description |
| User | Search Artist | Query the Class-Library |
| Search Song | Query the Class-Library |
| Enter IDs | IDs to be used for Comparison |
| Choose Metrics | Metric choice to compare features |
| Artist v Artist Dictionary Creation | Invoke the function |
| Rate Music | Convey liking of Music (not included) |
| Admin | Calculate Similarity | Invoke the chosen function |
| Show Predictions | Recommendation Scores |
| Gather Search History Data | Log user activities (search history) |

### Class Diagram

### Pseudocode

#### Load dataset module

#### Similarity module

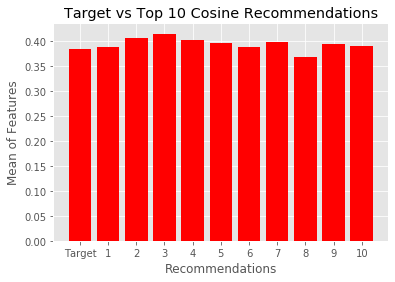
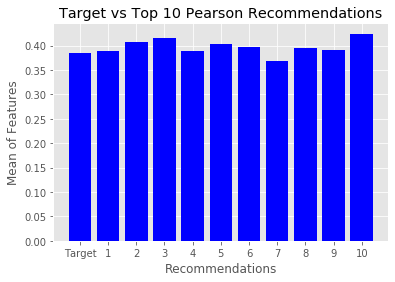
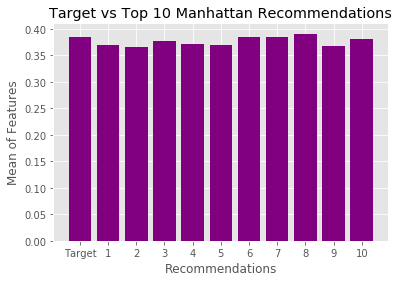
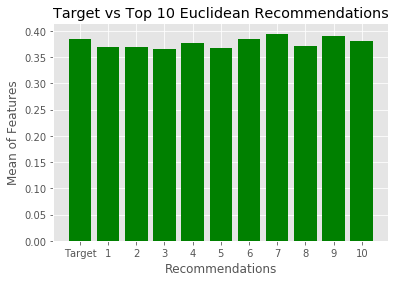
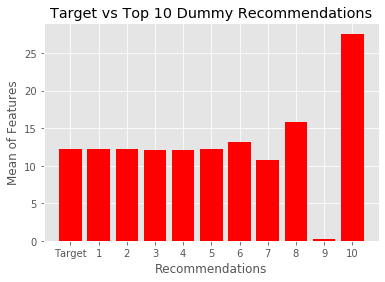
#### Main Notebook

### Requirement Testing Matrix

TODO-refine to match Assignment 2

|  |  |  |
| --- | --- | --- |
| **Requirement** | **Use Case** | **Response** |
| User can choose artist or song | Enter artist and song | Program proceeds |
| Enter invalid entry | Program complains and asks if user wants to quit |
| Artist Searching | Does user want to compare?  User enters ‘yes’ | Program continues, starts artist comparison |
| Does user want to compare?  User enters ‘no’ | Program continues, prompts the user regarding artist searching |
| User enters a correct name | Program prints search results, prompt for the user |
| User enters an invalid name | Program prints that there no results |
| User enters an invalid entry | Program prints a message, starts the metric compare section |
| Song Searching | User wants to search for a song | Song searching begins |
| User enters an invalid entry | Program restarts |
| User enters a word | Searching finds songs with the matching word |
| User enters multiple words | Searching finds songs with any of the entered words  **Needs refinement!** |
| User enters numbers or symbols | Searching returns matches |
| User enters ‘no’ | Metric selection begins |
| Metric Selection | User enters a correct number from the list provided | Expected Metric function runs |
| User enters a character | Program restarts |
| User enters an invalid number | Program states entry is wrong, asks the user to retry |
| Entering IDs for features | User enters two correct IDs  These IDs match | Program complains, asks the user to enter two IDs again as the IDs can’t match |
| User enters two correct IDs  These IDs don’t match | Program asks for the feature |
| User enters a character | Program prints an error message and ends, **could be refined** |
| User enters a number bigger than the size of the list | Program says the feature is wrong, could refine the error message, but functionality is as expected |
| Feature Input | User enters a valid feature from the list provided | Output the similarity score |
| User enters an invalid feature, number or otherwise | Program tells the user that the feature doesn’t exist |
| User enters ‘no’ or nothing | Output all features similarity scores |
|  |  |
|  |  |
|  |  |

### Evaluation Plots

Plots were created using Target ID 76110

